Manycore Network Interfaces for In-Memory Rack-Scale Computing

Manycore Network Interfaces for In-Memory Rack-Scale Computing

Alexandros Daglis Stanko Novaković Edouard Bugnion Babak Falsafi Boris Grot†
EcoCloud, EPFL †University of Edinburgh
{alexandros.daglis,stanko.novakovic,edouard.bugnion,babak.falsafi}@epfl.ch
†boris.grot@ed.ac.uk

Abstract

Datacenter operators rely on low-cost, high-density technologies to maximize throughput for data-intensive services with tight tail latencies. In-memory rack-scale computing is emerging as a promising paradigm in scale-out datacenters capitalizing on commodity SoCs, low-latency and high-bandwidth communication fabrics and a remote memory access model to enable aggregation of a rack's memory for critical data-intensive applications such as graph processing or key-value stores. Low latency and high bandwidth not only dictate eliminating communication bottlenecks in the software protocols and off-chip fabrics but also a careful on-chip integration of network interfaces. The latter is a key challenge especially in architectures with RDMA-inspired one-sided operations that aim to achieve low latency and high bandwidth through on-chip Network Interface (NI) support. This paper proposes and evaluates network interface architectures for tiled manycore SoCs for in-memory rack-scale computing. Our results indicate that a careful splitting of NI functionality per chip tile and at the chip's edge along a NOC dimension enables a rack-scale architecture to optimize for both latency and bandwidth. Our best manycore NI architecture achieves latencies within 3% of an idealized hardware NUMA and efficiently uses the full bisection bandwidth of the NOC, without changing the on-chip coherence protocol or the core's microarchitecture.

1. Introduction

The information revolution of the last decade has been fueled by rapid growth in digital data. IDC estimates a 300-fold increase in the size of the digital universe in the span of 15 years, totaling over 40 zettabytes by the year 2020 and doubling every 18 months [17]. Such rapid growth in data volumes has challenged datacenter operators that host latency-sensitive online services.

Existing datacenters are built in a scale-out fashion using commodity servers that provide cost efficiency and expansion friendliness. However, a number of emerging application classes, such as those that rely on graph-based data representations, are not directly amenable to sharding and intrinsically require inter-node transfers once the dataset size exceeds the memory of a single node [13, 35]. In contrast, cache-coherent NUMA machines seamlessly scale the memory capacity and latency with the number of sockets, but are notoriously difficult to scale to large configurations, are expensive to acquire, and present a fault-containment challenge [11].

Driven by these observations, researchers and system vendors have sought to coarsen the basic unit of compute in a way that would afford low-latency and high-bandwidth access to large amounts of memory without losing the benefits of the scale-out deployment model. As a result, *rack-scale computing* has emerged as a promising paradigm that combines direct remote memory access technology with lossless integrated fabrics and light-weight messaging in a rack-scale form factor [4, 38, 39]. A recent rack-scale system proposal demonstrated remote memory access latency of 300ns and ability to stream at full memory bandwidth of the remote socket [38].

Given the server technology trends, rack-scale systems will soon feature ARM-based SoCs with dozens of cores (e.g., Scale-Out Processors [10, 34] or Tiled Manycores [14]), high-bandwidth memory interfaces supplying well over 100GBps of DRAM bandwidth per socket, and SerDes or photonic chip-to-chip links, allowing for low-latency and high-bandwidth intra-rack communication. A rack-scale system features many such servers, tightly integrated in a supercomputer-like fabric. A key emerging challenge in such systems is a manycore SoC network interface (NI) architecture that would enable effective integration of on-chip resources with supercomputer-like off-chip communication fabrics to maximize efficiency and minimize cost.

Many have proposed on-chip NIs including designs integrated into the cache hierarchy of lean SoCs with one [38] or a few cores [26] by simply placing the NIs at the edge of the chip. Unfortunately, placing NIs at the chip's edge in a manycore SoC incurs prohibitively high on-chip coherence and NOC latencies on accesses to the NI's internal structures. As this work shows, on-chip latency is particularly high (up to 80% of the end-to-end latency) for fine-grain (e.g., cache block size) accesses to remote in-memory objects. Because of the demand for low remote access latency in rack-scale systems, such edge-based NI placements are not desirable.

Alternatively, there are manycore tiled processors with lean per-tile NIs directly integrated into the core's pipeline [7]. While per-tile designs optimize for low latency, they primarily target fine-grain (e.g., scalar) communication and are not suitable for in-memory rack-scale computing with coarse-grain objects from hundreds of bytes to tens of kilobytes. Moreover,

current per-tile designs are highly intrusive in microarchitecture, which is undesirable for licensed IP blocks (e.g., ARM cores) used across many products. Finally, these designs have primarily targeted single-chip systems rather than rack-scale systems, which rely on a remote memory access model.

This paper is, to the best of our knowledge, the first to evaluate the design space of manycore NIs for in-memory rack-scale systems. We study systems with RDMA-inspired one-sided operations (e.g., read/write) to enable low latency and high bandwidth for variable-size objects and make the following observations: (1) there is a need for per-tile NI functionality to eliminate unnecessary coherence traffic for fine-grain requestor-side operations, (2) given high coherence-related NOC latencies, the software overhead to trigger one-sided operations is amortized, obviating the need for direct remote load/store operations in hardware to accelerate them, (3) bulk transfer operations overwhelm the NOC resources and as such are best implemented at the chip's edge, and (4) response-side operations (i.e., remote requests to a SoC's local memory) do not interact with the cores and are therefore best handled at the chip's edge.

We use these observations and propose three manycore NI architectures: (1) NI_{edge}, the simplest design, with NIs along a dimension of the NOC at the chip's edge, optimizing for bandwidth and low on-chip traffic, (2) $\text{NI}_{per-tile}$, the most hardware-intensive design, with an NI at each tile to optimize for lower access latency from a core to NI internals, and (3) NI_{split}, a novel manycore NI architecture with a per-tile frontend requestor pipeline to initiate transfers, and a backend requestor pipeline for data handling plus a response pipeline servicing remote accesses to local memory, both integrated across the chip's edge. The NI_{split} design optimizes for both latency and bandwidth without requiring any modifications to the SoC's cache coherence protocol, the memory consistency model or the core microarchitecture.

We assume a 512-node 3D-torus-connected rack with 64-core mesh-based SoCs, and use cycle-accurate simulation to compare our three proposed manycore NI architectures to a NUMA machine of the same size and show that:

- On-chip coherence and NOC latency dominate end-to-end latency in manycore SoCs for in-memory rack-scale systems, amortizing the software overhead of one-sided operations. As such, intrusive core modifications to add hardware load/store support for remote operations are not merited;
- An NI_{edge} design can efficiently utilize the full bisection bandwidth of the NOC while incurring 16% to 80% end-to-end latency overhead as compared to NUMA;
- An $\text{NI}_{per-tile}$ design can achieve end-to-end latency within 3% of NUMA, but can only reach 25% of the bandwidth that NI_{edge} delivers for large (8KB) objects, due to extra on-chip traffic;
- An NI_{split} design combines the advantages of the two base designs and reaches within 3% of NUMA end-to-end latency, while matching NI_{edge}'s bandwidth.

2. Background

2.1. Application and Technology Trends

Today's networking technologies struggle to satisfy the heavy demands of datacenter applications that query and process massive amounts of data in real time. User data is growing faster than ever, and providing applications with fast access to it is fundamental. Because datasets commonly exceed the capacity of a single cache-coherent NUMA server, distributing data and computation across multiple servers (a.k.a., scale-out) has become the norm.

Unfortunately, most such applications must address large amounts of data in little time [6], with implications in terms of both latency and bandwidth. Many datacenter applications are hard to partition optimally as they rely on irregular data structures such as graphs, making the poor locality of reference a fact of life. Other applications, such as distributed key-value stores, force clients to go over the network in order to access just a few bytes of user data. Most key-value stores today operate on object sizes between 16 and 512 bytes, which are typical of datacenter applications [5, 43]. Similarly, Facebook's Memcached pools typically have objects close to 500 bytes in size [5]. Accesses to such small objects are bound by the network latency (up to 100μs), which can increase the overall latency as compared to local memory access latency ($<$100ns) by three or more orders of magnitude.

The corresponding bandwidth requirements are equally dramatic for datacenter applications. Lim et al. [32] measure object sizes in file servers, image servers and social networks varying in size from a few to tens of KBs. Many graph processing and MapReduce applications require more coarse-grained accesses and thus are mostly bound by the bisection bandwidth. In such applications, bandwidth requirements grow with the size of the system, as the fraction of data local to a given node is inversely proportional to the number of nodes.

Recently, server performance and energy considerations have led to the emergence of highly integrated chassis- and rack-scale systems, such as HP Moonshot [22], Boston Viridis [8], and AMD's SeaMicro [12]. These systems interconnect a large number of servers, each with an on-chip NI, using a supercomputer-like lossless fabric. NI integration and short intra-rack communication distances help reduce communication delays. Unfortunately, these benefits are offset by the deep network stacks of commodity network protocols. Moreover, the deep stacks' effect on performance is exacerbated by the lean cores in these servers.

Similarly, augmenting rack-scale systems with PCIe-attached RDMA controllers [21] can bring latency down to the microseconds range. However, remote access latency is still over 10x of local memory, limited by the PCIe bus and the delay of a complex RDMA-compliant ASIC. The PCIe bus also limits the bandwidth to 40Gbps (5GBps) in most configurations, which is grossly mismatched with local DRAM bandwidth that is approaching 20GBps per DDR channel.

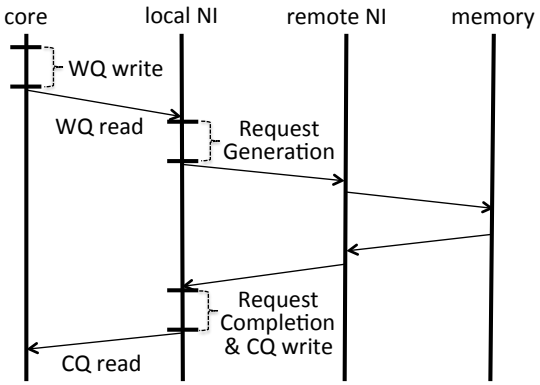


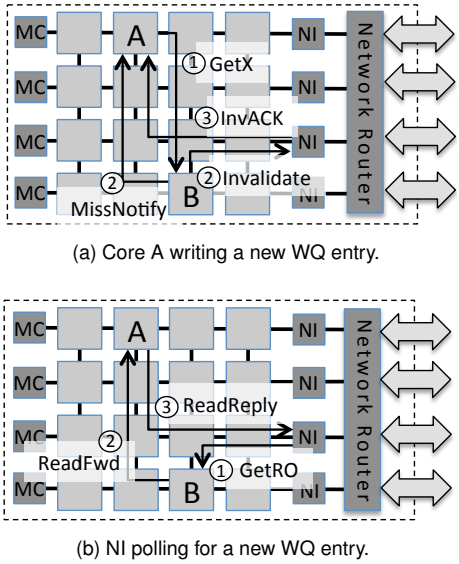
Figure 1: QP-based remote read.

This work focuses on low-latency and high-bandwidth remote memory access in rack-scale systems. We identify two attributes in tomorrow's servers that drive our design choices. First, we observe that research results and industry trajectory are pointing in the direction of server processors with dozens of cores per chip [10, 14, 34]. Thus, remote memory architectures will have to cope with realities of a *fat* node with many cores and non-trivial on-chip communication delays. Second, the emerging System-on-Chip (SoC) model for server processors favors features that can be packaged as separate IP blocks and eschews invasive modifications to existing IP (e.g., most licensees of ARM cores cannot afford an ARM architectural license that would allow them to modify core internals).

2.2. QP-based Remote Memory Access

Recent research [38] advocates in-memory rack-scale computing for low latency and high bandwidth using RDMA-inspired one-sided memory operations with architectural support in a specialized NI. In such implementations, the cores communicate with the NI via in-memory control structures, typically Queue Pairs (QPs), to schedule remote operations and get notified of their completion. The QP-based communication of RDMA introduces a non-trivial scheduling overhead for remote operations, as illustrated in Fig. 1. Each remote read or write requires the execution of multiple instructions on the core to create an entry in the Work Queue (WQ). The local NI polls on the WQ and upon the creation of a new request, the NI reads the corresponding WQ entry, generates a request and injects it into the network. Upon the response's arrival, the local NI takes protocol-specific actions to complete the request and notifies the application by writing to a Completion Queue (CQ). The QP-based approach is clearly more complex from the programming perspective than a pure load/store model, but is highly flexible and does not require modifying the core.

The underlying technology and mechanisms used to connect the controller to the cores play a significant role for the end-to-end latency and bandwidth. Most existing RDMA-based solutions rely on PCIe to connect the NI, introducing significant interaction overhead. While integrated on-chip solutions leverage cache coherence to make this interaction as cheap as possible, they still introduce non-negligible overhead due to sequences of coherence-related on-chip messages, triggered by each cache block transfer.



(a) Core A writing a new WQ entry.

(b) NI polling for a new WQ entry.

Figure 2: Core and NI WQ interactions.

QP-based communication at rack scale has the potential to drive remote access latency down to a small factor of DRAM latency. The QP-based approach is appealing because it does not require modifying the instruction set to support remote reads/writes. By using memory-mapped queues, applications can interact with the NI directly and bypass the OS kernel, which is a major source of overhead. A cache-coherent NI can provide applications with a remote access latency of just 300ns and the capability of streaming from remote memory at full DDR3 memory bandwidth [38]. However, current proposals are not applicable to manycore SoCs as they only evaluate single-core nodes. In this paper, we identify the challenges of NI placement and core-NI communication in a QP-based remote memory access model for manycore chips.

3. Manycore Network Interfaces

In this section, we investigate the design space for scalable network interfaces for manycore SoCs, their latency and bandwidth characteristics, and the costs and complexities associated with their integration.

3.1. Conventional Edge-Based NI

Recent high-density chassis- and rack-scale systems are based on server processors with a few cores and an on-chip NI [9, 12, 26]. Although integrated, the whole NI logic is still placed at the chip's edge, close to the pins that connect the chip to the network. We refer to this NI architecture as NI_{edge}.

Emerging scale-out server processors [34] (e.g., Cavium's ThunderX [10]), and tiled manycores (e.g., EZChip's TILE-Mx [14]) already feature from several dozens to up to 100 ARM cores. Because of this trend toward *fat* manycores,

NI_{edge} may not be optimal for chips that feature fast remote memory access powered by a QP-based model. In particular, the QP-related traffic between the cores and the NIs must traverse several hops on the NOC, and as the NOC grows with the chip's core count so does the number of hops to reach the chip's edge. Moreover, every cache block transfer triggers the coherence protocol, which typically requires several messages to complete a single transfer. Thus, the QP-related traffic (i.e., WQ/CQ read/write in Fig. 1) becomes a significant fraction of the end-to-end latency.

Fig. 2a and 2b illustrate the critical path for reads and writes, respectively, in an example manycore chip with NI_{edge}. Without loss of generality we assume a manycore chip with a mesh NOC and a statically block-interleaved shared NUCA LLC with a distributed directory and a 3-hop invalidation-based MESI coherence protocol. As such, a block's home tile location on the chip is only a function of its physical address. The NI also includes a small cache to hold the QP entries, which is integrated into the LLC's coherence domain [38] and is bypassed by all of the NI's data (non-QP) accesses.

Fig. 2a shows the sequence of coherence transactions required for core A to write to a WQ entry. The core sends a message to request an exclusive copy (*GetX*) of the cache block where the head WQ entry resides. The message goes to the requested block's home directory, which happens to be at core B ①. Because the NI polls on the WQ head, the directory subsequently invalidates NI's copy of the cache block, and concurrently sends the requested block to core A, notifying it (*MissNotify*) to wait for an invalidation acknowledgement from the NI ②. The NI then invalidates its copy of the requested block and sends an acknowledgement (*InvACK*) to resume core A ③. Once core A resumes, it also sends an acknowledgement concurrently to the directory to conclude the coherence transaction (not shown for brevity).

Fig. 2b shows the sequence of coherence transactions required for the NI to read a new WQ entry. The NI requests a read-only copy (*GetRO*) of the block from the directory ①. Because the block is modified in core A's cache, the directory sends a forward request to core A ②. Core A forwards a read-only copy of the modified block to the NI ③, downgrading its own copy, and resuming the NI. Once the NI resumes, it also sends an acknowledgement to the directory with a copy of the block to keep the data in the LLC up to date and conclude the coherence transaction (not shown for brevity).

The on-chip coherence overhead in terms of message count is similar in the case of CQ interactions. The only difference is that the roles of the core and the NI are reversed, with the NI writing entries in the CQ, and the core polling on the CQ head.

We now quantify these interactions through a case study. Table 1 presents a breakdown of the average end-to-end latency for a remote read operation under zero load in a QP-based rack-scale architecture featuring 64-core SoCs with a mesh NOC and NI_{edge}. In this breakdown, we assume communication between two directly connected chips (i.e., one network hop apart). The details of the modeled configuration can be found in §5. The table also includes the latency breakdown for a base NUMA machine (e.g., Cray T3D [27, 41]), which does not incur any QP-related on-chip communication overheads, as a point of comparison.

Latency Component	QP-based model	Latency Component	NUMA
A1) WQ write (core)	104	B1) Exec. of load instruction	1
A2) WQ read (NI)	95	B2) Transfer req. to chip edge	23
A3) Intra-rack network (1 hop)	70	B3) Intra-rack network (1 hop)	70
A4) Read data from memory	208	B4) Read data from memory	208
A5) Intra-rack network (1 hop)	70	B5) Intra-rack network (1 hop)	70
A6) CQ write (NI)	79	B6) Transfer reply to core	23
A7) CQ read (core)	84		
Total (2GHz cycles)	710	**Total (2GHz cycles)**	395
Overhead over NUMA	79.7%	-	

Table 1: Latency comparison of a QP-based model and a pure load/store interface.

Table 1 indicates that the overhead of the core writing a new WQ entry and the NI reading it can measure up to ∼200 cycles (entries A1 & A2), while the overhead for NUMA to send a request to the chip's edge is only 24 cycles (B1 & B2). The network and memory access at the remote node incur the same latency in both systems. Finally, the QP-based model requires ∼160 cycles to complete the transfer via a CQ entry that is written by the NI and read by the core (A6 & A7), while for NUMA the response is sent directly to the issuing core (B6).

The overall overhead of the QP-based model over a NUMA machine is almost 80%. The QP-based interactions that require multiple NOC transfers dominate the end-to-end latency. Moreover, in this example, the software overhead of creating a WQ entry for a RISC core is roughly a dozen arithmetic instructions plus two stores to the same cache block. Similarly, reading the CQ involves four instructions including a load. Therefore, the software overhead of reading/writing the QP structures is negligible compared to the overall on-chip latency. These results suggest that supporting a load/store hardware interface for remote accesses as in NUMA machines is an overkill because its impact would be negligible on the end-to-end latency for manycore chips.

NI_{edge} becomes competitive, however, with an increase in transfer size. The QP-based model allows for the core to issue a request for multi-cache-block objects through a single WQ entry. Such a request is subsequently parsed by the NI and "unrolled" directly in hardware, completing multiple block transfers before having to interact with the core again, thus amortizing the QP-related overheads over a larger data transfer. The net result is that NI_{edge} exhibits robust bandwidth characteristics with the QP-based remote access model. In contrast, a NUMA machine primarily supports a single cache block transfer and, without specialized NI support, it would suffer in performance from prohibitive on-chip traffic.

3.2. Per-Tile NI

An alternative NI design is to collocate the NI logic with each core ($NI_{per-tile}$, Fig. 3a). Such a design would mitigate QP-related traffic, thereby using the NOC only for the direct transfer of network packets between each tile and the network router at the chip's edge. To eliminate the coherence traffic between the core and NI, while precluding changes to the core or the LLC coherence controllers, the NI cache must be placed close to the core with care. We discuss the details of the NI cache design in §3.4.

Unfortunately, while $NI_{per-tile}$ minimizes the initiation latency for small transfers, it suffers from unnecessary NOC traversals for large transfers. To access a large object in remote memory, the NI issues a separate pair of request and response messages for each cache block. Thus, a large request is transformed into a stream of cache-block-sized requests, which congests the NOC on its way to the chip's edge. Similarly, the responses congest the NOC because every single response message must be routed to the NI, which the request originated from, before its payload is sent to its corresponding home LLC tile. Therefore, in contrast to NI_{edge}, $NI_{per-tile}$ optimizes for latency, while suffering from lower bandwidth. We compare and contrast the latency and bandwidth characteristics of these two NI designs in §6.

3.3. Split NI

To overcome the limitations of the NI_{edge} and $NI_{per-tile}$ designs, we propose a novel design that optimizes for both low latency and high bandwidth. Our design is based on the fundamental observation that an NI implements two distinct functionalities that are separable: (i) a *frontend* component including the NI cache, which interacts with the application to initiate a remote memory access operation, and (ii) a *backend* component, which accesses data. We therefore split each NI into these two components. We replicate the NI's frontend at each tile, so that each frontend is collocated with the core it is servicing to minimize the QP coherence overhead. The backend is replicated across the chip's edge, close to the network router. The split NI design (NI_{split}, Fig. 3b) achieves the best of both NI_{edge} and $NI_{per-tile}$ worlds. It provides low QP interaction latency without generating unnecessary NOC traffic and optimizes for both fine-grained and bulk transfers.

3.4. NI Cache

In the NI_{edge} design, each NI is attached to an edge tile, extending the mesh as shown in Fig. 2. Each NI includes a small cache that holds the QP entries and acts like a core's L1 data cache participating in the LLC's coherence activity. The NI cache has a unique on-chip tile ID, which is tracked by the coherence protocol much like a core's L1 cache.

In contrast, $NI_{per-tile}$ and NI_{split} collocate their NI cache with a core at each tile to mitigate coherence traffic induced by QP interactions. Unfortunately, a naive collocation of the

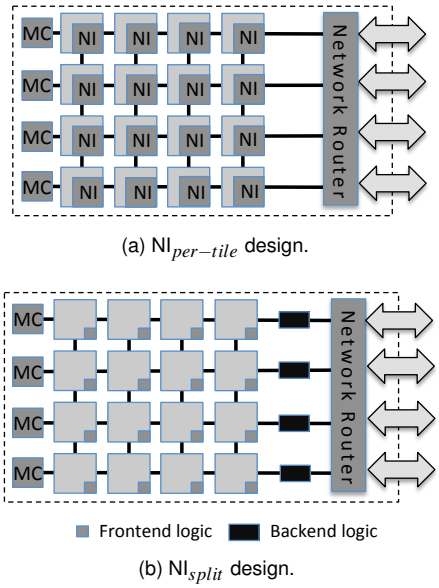


Figure 3: NI design space for manycore chips.

NI cache at each core does not eliminate the traffic because all QP interactions require consulting with the home directory in the LLC for the corresponding cache blocks. To guarantee that the traffic remains local, the system must migrate the home directories for the QP entries to their corresponding NI tile requiring additional architectural and OS support (e.g., as in R-NUCA [19]). Alternatively, sharing the L1 data cache between the core and NI would eliminate all traffic but is highly intrusive because the L1 data cache is on the critical path of the pipeline. Moreover, commodity ARM cores are licensed as an entire IP block and making changes to the core would be prohibitively expensive.

Instead, the cache for these two NI designs is attached directly to the back side of L1, at the boundary of the core's IP block. Unlike the NI_{edge} cache, this cache directly snoops all traffic from the L1's back side (e.g., as in a write-back or victim buffer). The NI cache and the core's L1 at each tile collectively appear as a single logical entity to the LLC's coherence domain while physically decoupled. Such an integration obviates the need to modify the on-chip coherence protocol and guarantees preserving the base memory consistency model (e.g., Reunion vocal/mute cache pair [44], FLASH/Typhoon block buffers [20, 28, 40]).

In this work, without loss of generality, we assume a non-inclusive MESI-based invalidation protocol with an inexact directory (i.e., non-notifying protocol). Exact directories are also implementable but would require a more sophisticated finite-state machine to guarantee that a single shared copy is tracked by the directory between the NI cache and the L1 data cache. For brevity, we omit the state transition diagram of the controller, and instead describe the basic principles of its operation.

Much like typical L1 back-side buffers, the $NI_{per-tile}$ (or NI_{split}) cache snoops all traffic from both L1 and the directory.

The cache can provide a block upon a miss in L1 as long as the request message conforms with the cache state. Otherwise, the request is forwarded to the directory. Similarly, if the directory requests a block that is currently shared by the cache, the cache acts on the message, forwards it to L1, waits for a response from L1, and responds back to the directory.

A frequent case that occurs under normal system operation is the $NI_{per-tile}$ (or NI_{split}) cache holding a block in the modified state because of a CQ write, and the core polling on that block, requesting a read-only copy. Under a MESI-based protocol, the cache cannot respond with a dirty block to a read-only request, so it would have to write it back to the LLC first. To optimize for this common case, we introduce an owned state, only visible to the NI cache controller. This way, the cache can directly forward a clean version of the requested block to L1, while keeping track of its modified state, so that the block eventually gets written back in the LLC upon its eviction.

4. A Case Study with Scale-Out NUMA

To illustrate the design points of the previous section, we use Scale-out NUMA (soNUMA), a state-of-the-art rack-scale architecture that features a QP-based model for remote memory access operations. We begin with a brief overview of soNUMA, and then materialize the discussed NI design approaches as a case study.

soNUMA is a programming model, architecture, and communication protocol for low-latency and high-bandwidth in-memory processing and data serving. An soNUMA cluster consists of multiple SoC server nodes connected by an external fabric. Nodes communicate with each other via remote read and write operations, similar to RDMA. The latter allows application processes to directly access the memory of an application running on another node at high speed using virtual memory, effectively creating a partitioned global address space across the cluster. Remote accesses spanning multiple cache blocks are unrolled into cache-block-sized requests at the source node. Prior evaluation results using cycle-accurate simulation of uniprocessor nodes show that soNUMA can effectively provide access to remote memory at only a 4x multiple of local DRAM latency [38].

4.1. soNUMA NI Overview

In soNUMA, the NI is an architecturally exposed block called the *Remote Memory Controller (RMC)*. Every request in soNUMA goes through three distinct stages: it is generated at the requesting node, serviced at a remote node, and completed upon its return to the requesting node. These three logical stages are handled by three independent RMC pipelines: the Request Generation Pipeline (RGP), the Request Completion Pipeline (RCP), and the Remote Request Processing Pipeline (RRPP). Request generations and completions are communicated between the cores and the RMC pipelines through a QP-based protocol with memory-mapped queues.

We now briefly describe the three pipelines; a more detailed description of their functionality can be found in [38].

Request Generation Pipeline. The RGP (Fig. 4a) periodically polls all the WQs that are registered with it to check for new enqueued requests. Each request is converted into one or more network packets, with large requests (i.e., those spanning multiple cache lines) unrolled by the RGP into a sequence of cache-block-sized transfers. For write requests, the RGP loads the write data from the local node's memory hierarchy prior to injecting the packet into the network router.

Request Completion Pipeline. The RCP (Fig. 4b) receives responses from the network, matches them to the original requests, stores the received data into the local node's memory (for reads only), and notifies the application upon each request's completion by writing in the appropriate CQ.

Remote Request Processing Pipeline. The RRPP is the simplest pipeline in terms of protocol processing complexity. It services incoming remote requests by reading or writing local memory and responding appropriately.

4.2. soNUMA NI Scaling and Placement

Given the functional overview of soNUMA's NI pipelines, we proceed to illustrate how they can be mapped to the different NI designs presented in §3.

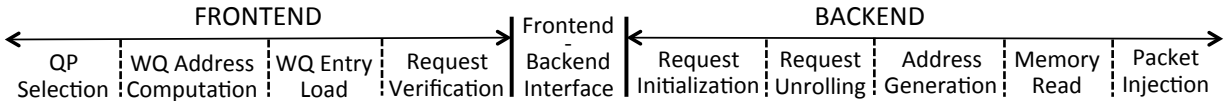
The RRPP pipeline is the only pipeline that does not interact with the cores. Therefore, in all the designs we consider in this work, it lies at the chip's edge nearest to the network router. In order to fully utilize the NOC bandwidth, multiple independent RRPPs are spread out along the edge (e.g., one per edge tile in a tiled CMP as shown in Fig. 2).

In an NI_{edge} design, the RGP/RCP scales like the RRPP – one pair per edge tile along a chip edge. In an $NI_{per-tile}$ design, a full RGP/RCP pair is replicated per tile and collocated with each core to minimize QP traffic. As described in §3.2, an essential requirement to reap the benefits of such a collocation is the proper integration of the NI cache with the chip's default coherence protocol and the core's L1 cache.

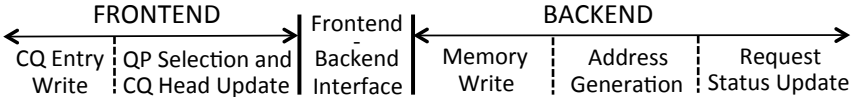
Both NI_{edge} and $NI_{per-tile}$ are suboptimal: the former latency-wise and the latter bandwidth-wise. We next discuss how to overcome the limitations of these designs in soNUMA with the NI_{split} design, which physically splits the RGP and RCP into a frontend and a backend.

RGP Frontend/Backend Separation. The frontend/backend split comes naturally in the RGP by separating the stages that interact with the WQs (frontend) from those that act on WQ requests by generating network packets (backend).

Fig. 4a details the functionality of RGP in stages. The RGP frontend selects a WQ among the registered QPs, computes the address of the target WQ, loads the WQ head, and checks if a new entry is present. The RGP backend initializes the NI's internal structures to track in-flight requests, unrolls large



(a) Request Generation Pipeline.



(b) Request Completion Pipeline.

Figure 4: Scale-Out NUMA RGP and RCP pipelines.

requests into cache-block-sized transactions, computes and translates the address of the target data and reads it from memory (for writes), and finally injects a request packet in the network.

In the figure, the *Frontend-Backend Interface* is the boundary between the frontend and the backend. In the NI_{edge} and $NI_{per-tile}$ designs, it is simply a pipeline latch. For the NI_{split} design, the *Frontend-Backend Interface* is an additional stage that generates and sends a NOC packet containing a valid WQ entry from the RGP frontend to its corresponding backend.

RCP Frontend/Backend Separation. The RCP frontend/backend split follows a similar separation of concerns as that of RGP. The RCP backend receives network packets and accesses local application memory to store the remote data. Once all of the response packets for a given request have been received, the frontend notifies the application of the request's completion by writing to the CQ.

Fig. 4b shows the RCP frontend and backend. The backend is responsible for updating the status of in-flight requests, computing the target virtual address at the local node, and storing the remote data at the translated address. The RCP frontend updates the CQ head pointer in RCP's internal bookkeeping structures and writes a new CQ entry at the CQ's head.

Similar to the RGP, the *Frontent-Backend Interface* is a latch in NI_{edge} and $NI_{per-tile}$ designs. For NI_{split}, it is an additional stage that packetizes and pushes a new CQ entry into the NOC from the RCP's backend to its corresponding frontend.

In the NI_{split} soNUMA design, we integrate the RGP and RCP frontends in each tile (Fig. 3b), thus minimizing the overhead of transferring the QP entries between the NI and the core. The interaction between the core and the frontend logic is handled through the mechanism described in §3.4. The RGP and RCP backends are replicated across the chip's edge nearest to the network router. Scaling the backend across the edge allows utilization of the NOC's full bisection bandwidth by locally generated requests.

4.3. Other Design Issues

Mapping of Frontends to Backends. There is no inherent limitation in the binding of a pipeline frontend to a backend. In this work, we consider a simple mapping, whereby all the frontends of a NOC row map to that row's backend, minimizing frontend-to-backend distance.

Mapping of Incoming Traffic to RRPPs. Distribution of incoming requests to the chip's RRPPs is address-interleaved to minimize the distance to the request's destination tile. This functionality can be trivially supported in the network router by inspecting a few bits of each request's offset field in its soNUMA header under the following assumptions: (i) the directory and LLC are statically address-interleaved across the chip's tiles, and (ii) the address bits that define a block's home location in the tiled LLC are part of the physical address (i.e., these bits fall within the page offset), so this location can be determined prior to translation. Such traffic distribution minimizes on-chip traffic and latency, as it guarantees a minimal number of on-chip hops for each request to reach its home location in the LLC.

On-chip Routing Implications. In our evaluated chip designs, we place NIs (RRPPs and RGP/RCP backends) on one side of the chip and memory controllers (MCs) on the opposite side. We found that on-chip routing is critical to effective bandwidth utilization, and conventional dimension-order routing, such as XY or even O1Turn [42], can severely throttle the peak data transfer bandwidth between the chip's NI and MC edges. This occurs because most packets originating at a remote node (i.e., remote requests as well as responses to this node's requests) end up as DRAM accesses, since the requested or delivered data is typically not found in on-chip caches. Under XY routing, all memory requests are first routed to the edge columns, where the MCs reside, and then turn to reach their target MC. The NOC column interfacing to the MCs turns into a bottleneck, reducing the overall bandwidth. If YX routing is used instead, a similar problem arises with responses originating at the MC tiles.

Recent work has proposed Class-based Deterministic Routing (CDR) [1] as a way of overcoming the MC column congestion bottleneck. CDR leverages both XY and YX routing, with the choice determined by the packet's message class (e.g., memory requests use YX routing while responses XY).

In the soNUMA design, the MC-oriented policy employed by CDR is insufficient, as edge-placed NIs (such as RGP/RCP backends in the case of NI_{split}) can also cause peripheral con-

gestion. To avoid the edge column with the NIs becoming a hotspot, we modify CDR by defining a new packet routing class for directory-sourced traffic; all messages of this class are routed YX, while the rest follow an XY route. This results in better utilization of the NOC's internal links and reduced pressure on the NOC's edge links, as directory-sourced traffic never turns at the chip's edges.

5. Methodology

Simulation. We use Flexus [46], a full-system cycle-accurate simulator, to evaluate our 64-core chip designs. The parameters used are summarized in Table 2. The NIs for all NI designs are modeled in full microarchitectural detail.

We focus our study on a single node, with remote ends emulated by a traffic generator that matches the outgoing request rate of the node that is simulated by generating incoming request traffic at the same rate. Incoming requests are address-interleaved among RRPPs as described in §4.3.

We assume a fixed chip-to-chip network latency of 35ns per hop [45] and monitor the average servicing latency of local RRPPs that are simulated in detail. This RRPP latency is added to the network latency (which is a function of hop count), thus providing the roundtrip latency of a request once it leaves the local node.

Interface Placement. We evaluate three different placements of the RGP and RCP NIs: NI_{edge}, $NI_{per-tile}$, and NI_{split}. For all three placements, RRPP NIs are placed across a chip's edge, next to the network router. This provides the ensemble of these NIs access to the full chip bisection bandwidth for servicing of incoming requests. Memory controllers are placed on the opposite side of the chip.

Memory and Network Bandwidth Assumptions. The focus of this work is the investigation of the implications of NI design on manycore chips. As such, we intentionally assume high-bandwidth off-chip interfaces to both memory and the intra-rack network that do not bottleneck our studied workloads. Technology-wise, high-bandwidth memory interfaces are emerging in the form of on-package DRAM [3] and high-speed SerDes. For instance, Micron's Hybrid Memory Cube provides 160GBps (15x more than a conventional DDR3 channel) with a quad of narrow SerDes-based links [25]. On the networking side, the recently finalized IEEE 802.3bj standard codifies a 100Gbps backplane Ethernet running over a quad-25Gbps interface, with Broadcom already announcing fully compliant 4x25Gbps PHYs. Beyond that, chip-to-chip photonics is nearing commercialization [24], with 100Gbps signaling rates demonstrated and 1Tbps anticipated [4].

Cores	ARM Cortex-A15-like; 64-bit, 2GHz, OoO, 3-wide dispatch/retirement, 60-entry ROB
L1 Caches	split I/D, 32KB 2-way, 64-byte blocks, 2 ports, 32 MSHRs, 3-cycle latency (tag+data)
LLC	Shared block-interleaved NUCA, 16MB total 16-way, 1 bank/tile, 6-cycle latency Mesh: 1 tile/core, NOC-Out: 8 tiles in total
Coherence	Directory-based Non-Inclusive MESI
Memory	50ns latency
Interconnect	16B links. 2D mesh: 3 cycles/hop NOC-Out: Flattened Butterfly: 2 tiles/cycle Tree Networks: 1 cycle/hop
NI	3 independent pipelines (RGP, RCP, RRPP) one RRPP per row (8 in total)
Network	Fixed 35ns latency per hop [45]

Table 2: System parameters for simulation on Flexus.

Network-on-chip. Since we do not throttle the network or memory bandwidth, the NOC becomes the main bandwidth limiter. We use a mesh as the baseline NOC topology and apply CDR to route on-chip traffic, as described in §4.3. We also validate the applicability of our observations on latency-optimized NOCs through a separate case study with NOC-Out [33], the state-of-the-art NOC for scale-out server chips.

Microbenchmarks. The goals of this work are to understand the implications of NI design choices on the latency and bandwidth of remote memory accesses in rack-scale systems. Toward that goal, our evaluation relies on microbenchmarks as a way of isolating software and hardware effects in tightly-integrated messaging architectures such as soNUMA while also facilitating a direct comparison to a hardware-only scheme (i.e., a NUMA architecture with a load/store interface to remote memory).

We use a remote read microbenchmark to measure the latency and bandwidth behavior of the evaluated NI designs. For latency, we study the unloaded case: a single core issuing synchronous remote read operations. Bandwidth studies are based on a remote read microbenchmark, in which remote reads are issued asynchronously: as long as there is space left in the WQ, the application keeps enqueueing new remote read requests, while occasionally polling its CQ for completions. If the 128-entry WQ is full, the application spins on the CQ until an earlier request's completion frees a WQ entry.

We vary the size of the remote reads from 64B to 16KB. Both the soNUMA context, the memory region accessed by remote requests, and local buffers, where requested remote data are written to, are sized to exceed the aggregate on-chip cache capacity, forcing all accesses to hit DRAM. We monitor the metrics of interest (latency, bandwidth) in 500K-cycle windows and run the simulation until the metric's value stabilizes (i.e., when the delta between consecutive monitoring windows is less than 1%).

6. Evaluation

6.1. Latency Characterization

We first provide a tomography of the end-to-end latency for a single block transfer and show where time goes for each of the three evaluated NI designs. We then show the latency sensitivity of a read request to the size of the transfer.

Latency Component	NI_edge	Latency Component	NI_per-tile	Latency Component	NI_split	Latency Component	NUMA projection
WQ write software overhead	104	WQ write software overhead	13	WQ write software overhead	13	Remote read issuing (single load)	1
WQ read and RGP processing	95	WQ entry transfer	5	WQ entry transfer	5		
		RGP Processing	7	RGP frontend processing	4		
		Transfer request to chip edge	23	Transfer request to RGP backend	23	Transfer request to chip edge	23
				RGP backend processing	4		
Intra-rack network (1 hop)	70	Intra-rack network (1 hop)	70	Intra-rack network (1 hop)	70	Intra-rack network (1 hop)	70
RRPP servicing	208	RRPP servicing	208	RRPP servicing	208	RRPP servicing	208
Intra-rack network (1 hop)	70	Intra-rack network (1 hop)	70	Intra-rack network (1 hop)	70	Intra-rack network (1 hop)	70
RCP processing and CQ entry write	79	Transfer reply to RCP	23	RCP backend processing	4	Transfer reply to requesting core	23
				Transfer reply to RCP frontend	23		
		RCP processing	11	RCP frontend processing	8		
		CQ entry transfer	5	CQ entry transfer	5		
CQ read software overhead	84	CQ read software overhead	10	CQ read software overhead	10		
Total (2GHz cycles)	710	**Total (2GHz cycles)**	445	**Total (2GHz cycles)**	447	**Total (2GHz cycles)**	395
Overhead over NUMA	79.7%	**Overhead over NUMA**	12.7%	**Overhead over NUMA**	13.2%	-	

Table 3: Zero-load latency breakdown of a single-block remote read.

6.1.1. Single-block Transfer Latency Breakdown

Table 3 shows the latency breakdown for a single-block remote read request. The first three design points show the performance for a messaging-based design, differing in the placement of the NIs that interact with the cores. The last column of the table is a projection of the performance of an ideal NUMA machine, which can access remote memory through its load/store interface without any of the overheads associated with messaging. We optimistically assume that issuing a load/store instruction only requires a single cycle. The cost of traversing the NOC from the core to the edge, network latency, and reading the data at the remote end are the same as for the messaging interface.

A critical observation is that the actual software overhead to issue and complete a remote read operation is mainly attributed to microarchitectural aspects rather than the number of instructions that need to be executed. While NI_{edge} suggests that the software overhead is as high as 188 cycles to issue and complete a request (the sum of *WQ write* and *CQ read* software overheads in Table 3), the other two designs show that the actual instruction execution overhead is just 23 cycles. The remaining 165 cycles are the result of bouncing a QP block between the core's and the NI's caches via the normal cache coherence mechanisms.

Although modern coherence mechanisms are considered to be extremely efficient for on-chip block transfers, these results indicate that high-performance NI designs should not rely on the assumption that coherence-powered transfers are free from a latency perspective. Coherence protocols intrinsically introduce points of indirection, which can turn a single transfer into a long-latency sequence of several multi-hop chip traversals. These subtle interactions must be taken into consideration when architecting a high-performance NI.

6.1.2. Scaling to More Network Hops

Fig. 5 projects the end-to-end latency for reading a single cache block in a rack-scale system, accounting for multiple network hops. The projection is based on Table 3's breakdown, accounting for 70 cycles (35ns) of network latency per hop, per direction. To put the numbers in perspective, the average and maximum hop counts between two nodes in a 512-node 3D torus deployment are 6 and 12 respectively.

Fig. 5 shows that the additional on-chip transfers related to QP interactions that occur in the case of NI_{edge} account for a significant fraction of the end-to-end latency, inducing a 28.6% overhead over NUMA for six network hops. In comparison, NI_{split} significantly reduces the time spent on QP interactions, bringing the end-to-end latency within 4.7% of NUMA. Even in the worst case of traversing the entire diameter of the modeled 3D torus, the difference in the end-to-end latency overhead between NI_{edge} and NI_{split} is still significant: 16.2% vs. 2.6% over NUMA. These results indicate that a high-performance NI design must consider the node's microarchitectural features, but highly

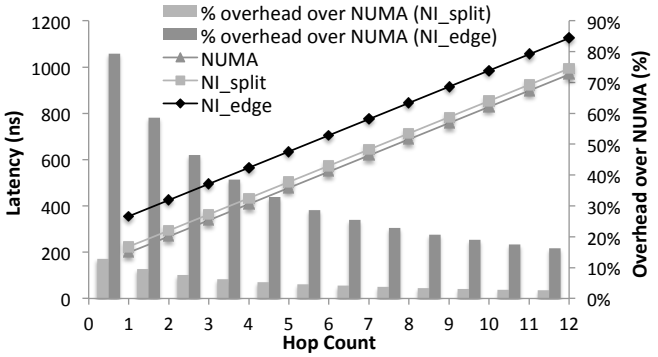


Figure 5: Projection of the end-to-end latency of a cache-block remote read operation for multiple intra-rack network hops. Bars map to the right y-axis, lines to the left y-axis.

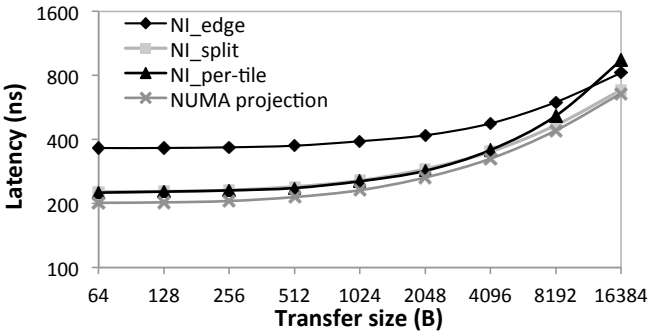


Figure 6: End-to-end latency for synchronous remote reads.

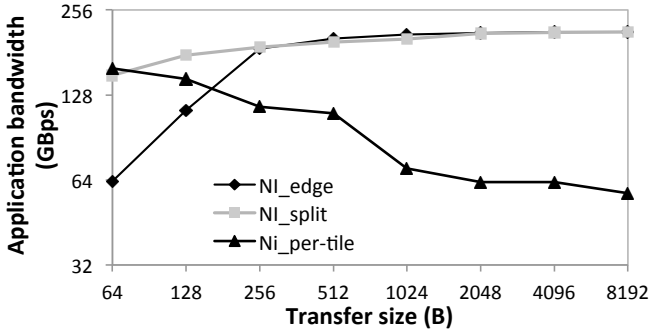


Figure 7: Application bandwidth for async remote reads.

invasive microarchitectural modifications (such as those associated with a load/store interface to remote memory) are not warranted.

6.1.3. Latency of Larger Requests

Fig. 6 shows the end-to-end latency of a synchronous remote read operation in an unloaded system assuming a single network hop per direction. We project the latency of an ideal NUMA machine by subtracting the latencies associated with QP interactions in the NI_{split} design as shown in Table 3.

We observe that as the transfer size increases, the relative latency difference between NI_{edge}, NI_{split}, and NUMA shrinks because the cost of launching remote requests through QP interactions is amortized over many cache blocks. However, that is not the case for $NI_{per-tile}$, which observes the highest latency among all evaluated designs for the largest transfer sizes. This behavior is caused by unrolls of large transfers into cache-block-sized transactions which, in the $NI_{per-tile}$ design, take place at the source tile. Because each network request packet is encapsulated inside a NOC packet, it requires two flits to transfer from the source tile to the network router at the chip's edge. Meanwhile, unrolls happen at a rate of one request per cycle, resulting in queuing at the source tile. While a wider NOC would alleviate the bandwidth pressure caused by unrolling at the source tile, the cost-effective solution is to provide hardware support for offloading bulk transfers to the chip's edge, as is done in the NI_{edge} and NI_{split} designs.

6.2. Bandwidth Characterization

For bandwidth measurements, we make all 64 cores issue asynchronous requests of varying sizes. Fig. 7 shows the aggregate application bandwidth for each of the three NI designs. The bandwidth is measured as the rate of data packets written into local buffers by RCPs for locally initiated requests, and the rate of data packets sent out by RRPPs in response to remote requests serviced at the local node. Because of the way remote traffic is generated, these two rates are always balanced, and the reported aggregate bandwidth is the sum of the two.

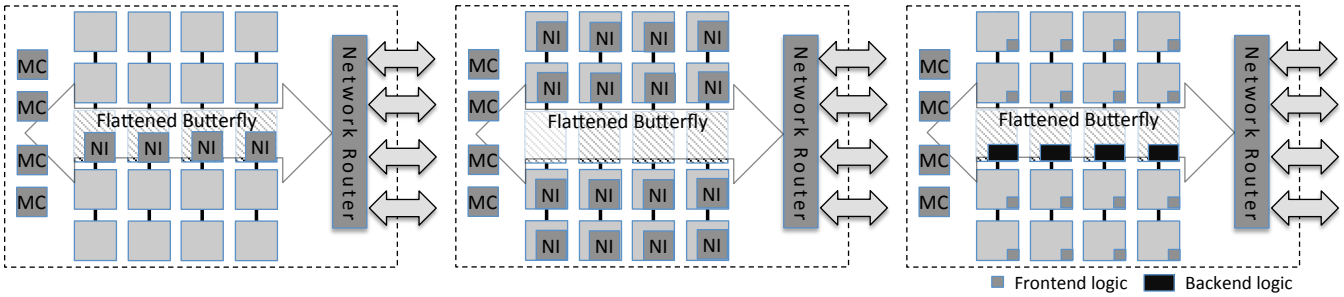
Both NI_{edge} and NI_{split} reach a peak bandwidth of 214GBps, or 107GBps per direction. It is unlikely that the bandwidth can be pushed any further using the same NOC; NOC traffic counters report an aggregate bandwidth of 594GBps, with the bulk of it crossing the bisection whose bidirectional bandwidth is 512GBps. The aggregate consumed bandwidth is 2.7x higher than the application bandwidth demand; the difference is attributed to a plethora of NOC packets that are not carrying application data. These other packets include coherence messages and evicted LLC blocks requiring a write-back to memory.

While we don't show this result, it is worth reporting that without CDR, the bandwidth curves are qualitatively similar, but the peak bandwidth any design can reach is less than half (~100GBps) of that achievable with CDR.

As Fig. 7 shows, $NI_{per-tile}$ and NI_{split} reach higher bandwidth than NI_{edge} for small transfer sizes. NI_{edge} suffers from ping-ponging of the WQ and CQ entries between the cores and the NIs, particularly when a cache block containing WQ entries gets polled and transferred to the NI before the application completely fills it with requests; a similar effect occurs with CQ cache blocks that are invalidated by the core while new completions are processed at the NI. With larger transfer sizes, QPs are accessed less frequently, thus diminishing their effect on performance.

Whereas NI_{edge} is inefficient for small transfers, the performance of the $NI_{per-tile}$ design degrades at large transfer granularities. The reason is that the NIs in this design unroll the requests inside the NOC, resulting in a flood of packets streaming from the tiles to the edges. By the time the backpressure reaches the source tiles, the network is completely congested. A similar problem occurs with responses: once they arrive at the network router, they are first sent back to the source NI, regardless of the final on-chip destination of the payload, thus introducing an unnecessary point of indirection, which further increases on-chip traffic. The congestion problems could be mitigated through smarter (e.g., source-based) flow-control, but the aggregate bandwidth would still be inferior to the other two designs because of the extra on-chip traffic due to the per-tile NI placement.

Clearly, efficient handling of large unrolls requires having a block handling engine at the edge, which receives a single command, does the data transfers, and finally notifies the requester upon completion. This observation is not limited to messaging, but equally applies to load/store NUMA systems



(a) NI$_{edge}$ design. (b) NI$_{per-tile}$ design. (c) NI$_{split}$ design.

Figure 8: NI design space for NOC-Out-based manycore CMPs. Striped rectangles represent LLC tiles.

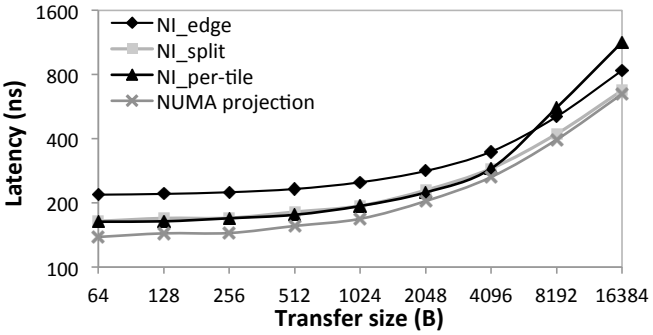


Figure 9: Latency for synchronous remote reads on NOC-Out.

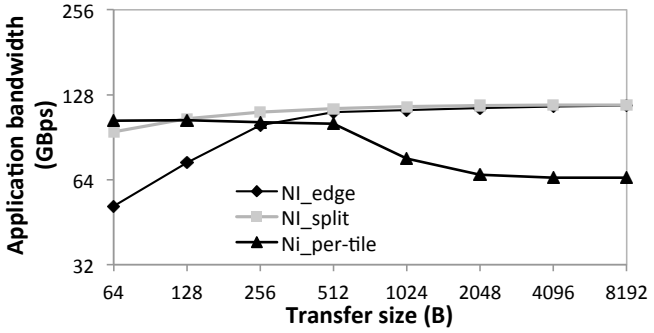


Figure 10: Application bandwidth for asynchronous remote reads on NOC-Out.

as well.

6.3. Effect of Latency-Optimized Topology

In this section, we show that trends and conclusions derived from the mesh-based study are equally valid for latency-optimized NOCs. To that end, we evaluate the various NI design options using NOC-Out [33], a state-of-the-art latency-optimized NOC for scale-out server chips. In the NOC-Out layout, LLC tiles form a row in the middle of the chip and are richly interconnected via a flattened butterfly. Cores lie on both sides of the LLC row, and the cores of each column are chained via a simple reduction/dispersion network that connects them to their column's corresponding LLC tile.

Fig. 8 illustrates the three NI design options in the context of NOC-Out. The LLC tiles are spread across the middle of the chip and are interconnected via the flattened butterfly to each other, the MCs, and the network router. In all three designs, the RRPPs (not shown) are placed across the chip's LLC tiles rather than the chip's edge, as the rich connectivity of these tiles provides access to the full bisection bandwidth. For the same reason, the RGPs and RCPs in the case of NI$_{edge}$ are collocated with the RRPPs. While *NI$_{middle}$* would be a more accurate term for this placement, we continue using *NI$_{edge}$* for consistency. NI$_{per-tile}$ features full RGP and RCP pipelines at each core, while NI$_{split}$ has an RGP/RCP frontend per core and an RGP/RCP backend per LLC tile.

6.3.1. Latency & Bandwidth Measurements

Fig. 9 shows the end-to-end latency of synchronous remote read operations of various sizes for all three NI designs. For small transfers, NOC-Out delivers up to 30% lower latency than mesh (Fig. 6). Examining the sources of improvement, we find that latency is reduced both at the source and remote nodes. Improvements at the source node originate from accelerated QP interactions and faster transfers of requests and responses between the NIs and the network router. At the remote node, the flattened butterfly speeds up the access latency to the LLC and MCs by 37% compared to mesh-based designs.

Comparing the latency gap between NI$_{edge}$ and the other two designs, we observe that it is narrowed compared to the mesh topology, yet the latency of NI$_{edge}$ is still up to 30% greater than that of NI$_{split}$ and NI$_{per-tile}$. This result indicates that on-chip QP interactions still account for a considerable fraction of the end-to-end latency even in latency-optimized NOC topologies.

Bandwidth results for NOC-Out appear in Fig. 10. The general trends are identical to those observed in the mesh (Fig. 7). However, the peak bandwidth achieved with NOC-Out is significantly lower than that in mesh-integrated NIs. The reason for the low throughput is the highly contended LLC in the NOC-Out organization, which has significantly fewer tiles and banks than its mesh-based counterpart.

7. Related Work

Some of the concepts and technologies mentioned in this work have been around for quite some time, while others are more recent. In this section, we focus on state-of-the-art NUMA solutions, remote access primitives, and coherent NIs.

NUMA. In the 90's, cache-coherent NUMA designs emerged as a promising approach to scale shared-memory multiprocessor performance by interconnecting thin symmetric multiprocessor server nodes with a low-latency and high-bandwidth network. These machines provided a globally coherent distributed memory abstraction to applications and the OS. Examples include academic prototypes such as Alewife [2], Dash [30], FLASH [20, 28], and Typhoon [40], and products such as SGI Origin [29] and Sun Wildfire [16, 18]. While most targeted coherence at cache block granularity, machines with programmable controllers also enabled support for bulk transfers [15, 20] broken down into a stream of multiple cache blocks. Today's multi-socket servers are cache-coherent NUMA machines with a few thin multicore sockets that use either Intel's QPI or AMD's HTX technology. In this work we conclude that moving to *fat* manycores has major implications on NI placement for both fine-grained and bulk transfers, which were previously not explored.

Remote Memory Access. Hardware support for remote access has been commercialized in Cray supercomputers [27, 41]. Cray T3D/T3E implemented *put* and *get* instructions that applications could use to directly access a global memory pool. Manycore NIs presented in this work focus on an RDMA-like programming model rather than a load/store model. Our results indicate that the software overhead of one-sided operations is likely to be a negligible fraction of end-to-end latency and as such hardware load/store interfaces would be an overkill. Modern RDMA-based NIs such as Mellanox ConnectX-3 [36] provide remote read and write primitives that applications can use to access remote memory via in-memory QPs. Most such adapters are PCIe-attached and therefore suffer from long latencies and low bandwidth.

Coherent NI. Coherent Network Interfaces (CNI) [37] use cacheable queues to minimize the latency between the NI and the processor. Such designs, however, do not assume large manycore chips where the NOC latency represents a significant fraction of the end-to-end latency. An NI can leverage coherent shared memory also to optimize TCP/IP stacks [7, 23, 31]. While previous proposals on NI optimization were focused on traditional networking and were thus inevitably engaged with expensive network protocol processing, our work focuses on specialized NIs for RDMA-like communication, in which execution of remote operations only requires low-cost user-level interactions with memory-mapped queues and minimal protocol processing.

8. Conclusion

The emergence of large manycore chips in the context of integrated rack-scale fabrics, where low latency and high bandwidth between nodes is crucial, introduces new challenges in the context of on-chip NI integration. Because of inherently high on-chip latencies, initiation and termination of remote operations that take place at the NIs can become a first-order performance determinant for remote memory access, especially for emerging QP-based models. This work investigated three different integrated NI designs for manycores. The classic NI_{edge} integration approach, where the NIs are placed across a chip's edge, can utilize the full bisection bandwidth of the NOC, but suffers from significant latency overheads due to costly on-chip core-NI interactions. The $\text{NI}_{per-tile}$ design integrates the NI logic next to each core, rather than the chip's edge, and delivers end-to-end latency for fine-grained remote memory accesses that is within 3% of hardware NUMA's latency. However, the $\text{NI}_{per-tile}$ design generates excess traffic that reduces the bandwidth for bulk data transfers significantly. To achieve the best of both worlds, this work proposed an optimized manycore NI design, NI_{split}, that delivers the latency of $\text{NI}_{per-tile}$ and the bandwidth of NI_{edge}.

Acknowledgements

The authors thank the anonymous reviewers for their precious comments and feedback. We thank Javier Picorel, Pejman Lotfi-Kamran, Stavros Volos and Sotiria Fytraki for fruitful technical discussions about NOCs and manycore coherence protocols, and Cansu Kaynak, Djordje Jevdjic, Nooshin Mirzadeh and the rest of the PARSA group for their feedback and support.

This work has been partially funded by the *Workloads and Server Architectures for Green Datacenters* project of the Swiss National Science Foundation, the Nano-Tera *YINS* project, and the *Scale-Out NUMA* project of the Microsoft-EPFL Joint Research Center.

References

[1] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti, "Achieving Predictable Performance Through Better Memory Controller Placement in Many-Core CMPs," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, 2009, pp. 451–461.

[2] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. A. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife Machine: Architecture and Performance," in *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*, 1995.

[3] Anandtech, "Haswell: Up to 128MB On-Package Cache." [Online]. Available: http://www.anandtech.com/show/6277/haswell-up-to-128mb-onpackage-cache-ulv-gpu-performance-estimates.

[4] K. Asanović, "A Hardware Building Block for 2020 Warehouse-Scale Computers," USENIX FAST Keynote, 2014.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-Scale Key-Value Store," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1, 2012, pp. 53–64.

[6] L. A. Barroso, "Three Things to Save the Datacenter," ISSCC Keynote, 2014. [Online]. Available: http://www.theregister.co.uk/Print/2014/02/11/google_research_three_things_that_must_be_done_to_save_the_data_center_of_the_future/.

[7] N. L. Binkert, A. G. Saidi, and S. K. Reinhardt, "Integrated Network Interfaces for High-Bandwidth TCP/IP," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[8] Boston Limited, "Boston Limited Unveil Their Revolutionary Boston Viridis," 2011. [Online]. Available: http://www.boston.co.uk/press/2011/11/boston-limited-unveil-their-revolutionary-boston-viridis.aspx.

[9] Calxeda Inc., "ECX-1000 Technical Specifications," 2012. [Online]. Available: http://www.calxeda.com/ecx-1000-techspecs/.

[10] Cavium Networks, "Cavium Announces Availability of ThunderX™: Industry's First 48 Core Family of ARMv8 Workload Optimized Processors for Next Generation Data Center & Cloud Infrastructure," 2014. [Online]. Available: http://www.cavium.com/newsevents-Cavium-Announces-Availability-of-ThunderX.html.

[11] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta, "Hive: Fault Containment for Shared-Memory Multiprocessors," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[12] A. Dhodapkar, G. Lauterbach, S. Li, D. Mallick, J. Bauman, S. Kanthadai, T. Kuzuhara, G. S. M. Xu, and C. Zhang, "SeaMicro SM10000-64 Server: Building Datacenter Servers Using Cell Phone Chips," in *Proceedings of the 23rd IEEE HotChips Symposium*, 2011.

[13] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "FaRM: Fast Remote Memory," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[14] EZchip Semiconductor Ltd., "EZchip Introduces TILE-Mx100 World's Highest Core-Count ARM Processor Optimized for High-Performance Networking Applications," Press Release, 2015. [Online]. Available: http://www.tilera.com/News/PressRelease/?ezchip=97.

[15] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood, "Application-Specific Protocols for User-Level Shared Memory," in *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing (SC)*, 1994.

[16] B. Falsafi and D. A. Wood, "Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA," in *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, 1997.

[17] J. Gantz and D. Reinsel, "The Digital Universe in 2020." IDC, 2012. [Online]. Available: http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf.

[18] E. Hagersten and M. Koster, "Wildfire: A scalable path for smps," in *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture (HPCA)*, 1999.

[19] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," in *36th International Symposium on Computer Architecture (ISCA)*, 2009.

[20] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta, "Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor," in *ACM SIGPLAN Notices*, vol. 29, no. 11, 1994, pp. 38–50.

[21] Hewlett - Packard Development Company, "HP ProLiant m400 Server Cartridge," 2014. [Online]. Available: http://www8.hp.com/h20195/v2/GetDocument.aspx?docname=c04384048.

[22] Hewlett-Packard Development Company, "HP Moonshot System Family Guide," 2014. [Online]. Available: http://www8.hp.com/h20195/v2/GetDocument.aspx?docname=4AA4-6076ENW.

[23] R. Huggahalli, R. Iyer, and S. Tetrick, "Direct Cache Access for High Bandwidth Network I/O," in *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, 2005.

[24] Intel, "Moving Data with Silicon and Light," 2013. [Online]. Available: http://www.intel.com/content/www/us/en/research/intel-labs-silicon-photonics-research.html.

[25] J. Jeddeloh and B. Keeth, "Hybrid Memory Cube New DRAM Architecture Increases Density and Performance," in *2012 International Symposium on VLSI Technology (VLSIT)*, 2012.

[26] D. Kanter, "X-Gene 2 Aims Above Microservers," *Microprocessor Report*, vol. 28(9), pp. 20–24, 2014.

[27] R. Kessler and J. Schwarzmeier, "Cray T3D: A New Dimension for Cray Research," in *Compcon Spring '93, Digest of Papers*, 1993.

[28] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy, "The Stanford FLASH Multiprocessor," in *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, 1994.

[29] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," in *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, 1997, pp. 241–251.

[30] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. L. Hennessy, M. Horowitz, and M. S. Lam, "The Stanford Dash Multiprocessor," *IEEE Computer*, vol. 25, no. 3, pp. 63–79, 1992.

[31] G. Liao, X. Zhu, and L. Bnuyan, "A New Server I/O Architecture for High Speed Networks," in *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[32] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[33] P. Lotfi-Kamran, B. Grot, and B. Falsafi, "NOC-Out: Microarchitecting a Scale-Out Processor," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[34] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, Y. O. Koçberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Özer, and B. Falsafi, "Scale-Out Processors," in *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)*, 2012.

[35] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2010.

[36] Mellanox Corp., "ConnectX-3 Pro Product Brief," 2012. [Online]. Available: http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-3_Pro_Card_EN.pdf.

[37] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood, "Coherent Network Interfaces for Fine-Grain Communication," in *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, 1996.

[38] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-Out NUMA," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[39] D. N. Paolo Costa, Hitesh Ballani, "Rethinking the Network Stack for Rack-Scale Computers," in *Hot Topics in Cloud Computing (HotCloud)*. USENIX, 2014.

[40] S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Tempest and Typhoon: User-Level Shared Memory," in *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, 1994.

[41] S. L. Scott and G. M. Thorson, "The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus," in *Hot Interconnects*, 1996.

[42] D. Seo, A. Ali, W.-T. Lim, N. Rafique, and M. Thottethodi, "Near-Optimal Worst-Case Throughput Routing for Two-Dimensional Mesh Networks," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005.

[43] W. Shi, E. Collins, and V. Karamcheti, "Modeling Object Characteristics of Dynamic Web Content," *Journal of Parallel and Distributed Computing*, vol. 63, no. 10, pp. 963–980, 2003.

[44] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe, "Reunion: Complexity-Effective Multicore Redundancy," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[45] B. Towles, J. Grossman, B. Greskamp, and D. E. Shaw, "Unifying On-Chip and Inter-Node Switching within the Anton 2 Network," in *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, 2014.

[46] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "SimFlex: Statistical Sampling of Computer System Simulation," *IEEE Micro*, vol. 26, pp. 18 –31, 2006.